

Testes e Estratégias de Depuração

Luiz Henrique de Campos Merschmann
Departamento de Computação Aplicada
Universidade Federal de Lavras

luiz.hcm@ufla.br



Na Aula de Hoje



Teste e Depuração:

- Tipos de Erros
- Testes
- Estratégias de Depuração

Testes e Depuração

A implementação de uma classe é **raramente perfeita** depois da **primeira tentativa** de escrever seu código fonte. Normalmente, a **princípio não funciona corretamente** e mais trabalho é exigido para se fazer os acertos.



Tanto **programadores** iniciantes como experientes **lidam com erros** ao realizarem uma implementação.

Então precisamos **entender os tipos de erros** possíveis e as **formas de lidar** com os mesmos.

Tipos de Erros

Quando estamos programando, nem sempre as coisas funcionam como gostaríamos.

Normalmente, ao longo do processo de implementação de um código nos deparamos com vários erros.

Existem dois tipos de erros:

- ▶ **Erros de sintaxe:** quando o código viola as regras da linguagem, ou seja, sua sintaxe.
 - ▶ Esses erros são fáceis de achar, pois são detectados e apontados pelo compilador.
- ▶ **Erros de lógica:** são erros no código que fazem com que ele produza resultados errôneos quando executado.
 - ▶ Não são detectados pelo compilador e, portanto, não impedem que o código compile com sucesso.
 - ▶ Por isso, são mais difíceis de serem encontrados!

Teste e Depuração

A maioria dos softwares comerciais possui *bugs*! Como melhorar a qualidade de um software?

- ▶ A solução está nas atividades de **teste** e **depuração** do código.
- ▶ **Testar** significa **verificar** se uma parte do código faz o que deveria ser feito.
- ▶ **Depurar** (ou *debugar*) é o processo de **encontrar** a causa de um erro e **corrigi-lo**.
- ▶ A **depuração vem depois do teste**, ou seja, se os testes apontarem que existem erros, técnicas de depuração são utilizadas para localizá-los e corrigi-los.



Teste e Depuração

Existem várias técnicas de teste e depuração. É importante conhecê-las e saber quando usar cada uma.

Os testes podem ser:

► **Testes de sistema:**

- Refere-se ao teste da **aplicação completa**, como usuário final, para verificar se ela se comporta conforme o esperado.
- Existem várias técnicas de testes de sistema, mas elas não são o foco dessa disciplina.

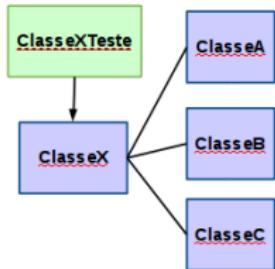
► **Testes de unidade:**

- Refere-se ao teste de **partes individuais** de uma aplicação, para verificar se aquela parte contém erros.
- As unidades testadas podem ser de diversos tamanhos: um grupo de classes, uma única classe ou mesmo um único método.
- Podem ser feitos manualmente ou de forma automatizada. Abordaremos a forma manual.



Teste de Unidade

Teste manual de uma classe



- ▶ Considere um sistema com muitas classes.
- ▶ Considere também que você implementou uma dessas classes (ClasseX) e que ela se relaciona com várias outras.
- ▶ Como podemos fazer para testar essa ClasseX?

Criando uma classe auxiliar ClasseXTeste!

- ▶ Essa classe auxiliar deve ser capaz de permitir a execução do código, ou seja, ela deve conter o método main.
- ▶ A ideia é que ela seja usada para instanciar objetos da ClasseX e chamar todos os métodos públicos da mesma.
- ▶ Vale ressaltar que essa ClasseXTeste não fará parte do sistema final, ou seja, ela serve apenas para fazer um teste completo da ClasseX.

Exemplo de Classe de Teste de Unidade

Esse é um trecho do código da classe que testa a classe ContaBancaria.

```
public class ContaBancariaTeste
{
    public static void main(String[] args)
    {
        ContaBancaria conta1 = new ContaBancaria(1234, "Luiz Henrique");

        System.out.println("Testando o número da conta...");
        if(conta1.getNumero() == 1234){
            System.out.println("[OK]");
        }
        else{
            System.out.println("[ERRO] Esperado: 1234, Retornado: " + conta1.getNumero());
        }

        System.out.println("Testando o saldo inicial da conta...");
        if(conta1.getSaldo() == 0){
            System.out.println("[OK]");
        }
        else{
            System.out.println("[ERRO] Esperado: 0, Retornado: " + conta1.getSaldo());
        }

        System.out.println("Testando o saldo após depósito de R$50...");
        conta1.depositar50();
        if(conta1.getSaldo() == 50){
            System.out.println("[OK]");
        }
        else{
            System.out.println("[ERRO] Esperado: 50, Retornado: " + conta1.getSaldo());
        }
    }
}
```

A ideia é criar objetos da classe ContaBancaria e verificar se seu estado se mantém coerente após cada chamada de método público.

Com essa classe implementada, após qualquer alteração na classe ContaBancaria você pode executar este programa novamente para verificar se tudo continua funcionando corretamente.

Testes

Teste manual

- ▶ O teste individual de classes durante o desenvolvimento do sistema diminui drasticamente as chances de o sistema como um todo apresentar problemas futuros.
- ▶ Portanto, ele é essencial em grandes projetos. Lembre-se que os prejuízos causados pelo atraso de projetos podem ser muito grandes.

Teste Automatizado

- ▶ Possuem a mesma ideia dos testes manuais, mas podem ser executados de forma automática (através de ferramentas de teste) evitando que você tenha que executar cada programa de teste de classe manualmente.

Estratégias de Depuração



Imagine que ao testar o programa ele apresenta um resultado diferente do que era esperado.

Qual é a primeira coisa que pensamos em fazer?

Nossa **primeira** estratégia de depuração é **ler o código** para tentar identificar porque a lógica dele está errada.

- ▶ Essa estratégia geralmente permite encontrarmos o problema em **sistemas** relativamente **pequenos e simples**.

Estratégias de Depuração

E quando o problema **não** é encontrado apenas **com a leitura do código?**



Nesse caso podemos partir para a segunda estratégia: **teste de mesa**.

O que é isso?

- ▶ O **teste de mesa** nada mais é do que montar uma **tabela** com **chamadas de métodos**, os **parâmetros e variáveis**, e colocar os valores que eles vão assumindo **a cada passo** da execução do algoritmo.

Método	ParamA	VariavelX	VariavelY
adiciona	10	true	18
adiciona	10	false	15

- ▶ Ou seja, vamos lendo o código passo a passo e executando “manualmente” o código de maneira a tentar identificar onde está a falha de lógica.

Estratégias de Depuração

Outra maneira de realizar uma verificação passo a passo é explicar para alguém o seu código.



Essa terceira estratégia de depuração é simples e pode parecer boba, mas pode ser muito eficaz na prática.

Por que ela funciona?

- ▶ A pessoa para quem você explica o código pode descobrir o erro para você.
- ▶ Mas frequentemente o simples ato de expressar em palavras o que o código faz nos impõe uma organização de pensamento que nos faz perceber o que está errado no código.
- ▶ Portanto, não subestime essa alternativa de depuração!

Estratégias de Depuração

A quarta estratégia é utilizar instruções de impressão ao longo do código.



Essa estratégia é simples porque não depende de nenhum recurso especial da linguagem e nem de uma IDE específica. Portanto, ela pode ser facilmente utilizada em qualquer situação.

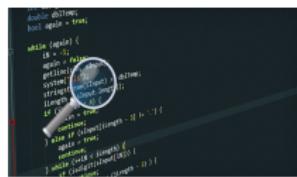
A ideia é imprimir informações que nos permitam acompanhar:

- ▶ A sequência de chamadas dos métodos.
- ▶ Os valores dos parâmetros.
- ▶ Os valores de variáveis locais e atributos em pontos estratégicos.

Lembre-se que essas impressões **não devem aparecer** no software final.

Estratégias de Depuração

A quinta estratégia de depuração é utilizar o depurador (debugger) da IDE.



O depurador é essencialmente uma ferramenta de software que fornece suporte para realizarmos uma verificação passo a passo em um segmento de código.

A partir da escolha de um ponto no código (*breakpoint*) no qual a execução será pausada, o depurador nos permitirá:

- ▶ Ver o estado dos objetos e os valores das variáveis.
- ▶ Executar os próximos comandos do código passo a passo.
 - ▶ Isso nos permitirá acompanhar as mudanças nos estados dos objetos e nos valores das variáveis.
- ▶ Ver a pilha de execução, isto é, a sequência de chamadas realizadas até aquele ponto.

Estratégias de Depuração

Depurador do BlueJ

Pilha de Execução

Aqui podemos visualizar os estados dos objetos, os valores dos parâmetros e das variáveis locais.

The screenshot shows the BlueJ debugger window with several key components highlighted:

- Pilha de Execução (Execution Stack):** On the left, it shows the call stack: main (no breakpoint) calling Prova.aplicar, which calls ProvaTeste.main. A red arrow points to the first line of code in the main method.
- Variáveis de classe (Class Variables):** A red box highlights this section, which contains private variables: scanner (Scanner), questao (Questao[]), and correcao (Correcao[]).
- Variáveis de instância (Instance Variables):** This section contains the current instance variables: entrada (Scanner), questao (Questao[]), and correcao (Correcao[]).
- Variáveis locais (Local Variables):** This section contains local variables: i (int), acertou (boolean), numeroTentativas (int), and resposta (int).
- Buttons at the bottom:** Parar (Stop), Avançar (Step Over), Entrar em (Step Into), Continuar (Continue), and Terminar (Terminate).

Breakpoint: A red arrow points to the line of code where a breakpoint is set: `numeroTentativas++;`

```
numeroTentativas = 0;
//Imprimindo uma questão
System.out.printf("Questão %d: %s", numeroTentativas + 1, questao[0].pergunta);
//Resposta da questão
resposta = entrada.nextInt();
//Armazenando a resposta
questao[0].correcao[0] = new Correcao(resposta);
//Verificando se a resposta é correta
if (questao[0].acertou == resposta) {
    System.out.println("Acertou!");
    acertou = true;
    break;
} else {
    System.out.println("Errou! A resposta correta é: " + questao[0].correcao[0].resposta);
    //Armazenando a resposta
    resposta = entrada.nextInt();
}
//Armazenando a correção
correcao[i] = new Correcao(resposta);

public void gerarRelatorio() {
    if (correcao[0] == null) {
        System.out.println("Nenhum resultado gerado.");
    } else {
        System.out.println("Relatório:");
        for(int i = 0; i < 5; i++) {
            System.out.printf("Pergunta %d: %s\n", i + 1, questao[i].pergunta);
            System.out.printf("Resposta: %d\n", resposta[i]);
            System.out.printf("Correção: %d\n", correcao[i].resposta);
        }
    }
}
```

Estratégias de Depuração

Depurador do BlueJ

Etapas para usar o **depurador** do *BlueJ*:

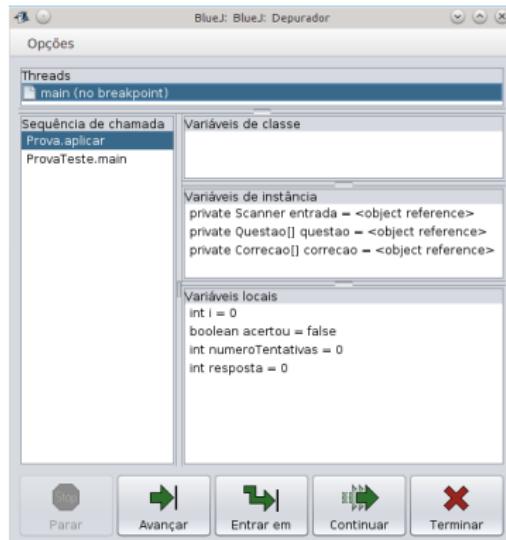
- **Abra o código-fonte** da classe onde deseja depurar.
- **Clique na linha** onde deseja **inserir** o **breakpoint**.
- **Execute o programa**.
 - Quando a execução alcançar a linha do **breakpoint**, o *BlueJ* irá **pausar** a execução e exibir a **janela de depuração** mostrada na figura ao lado.

Nessa janela de depuração, você poderá **visualizar a pilha de execução**, os **valores das variáveis** e dos **atributos** enquanto executa uma das ações a seguir:

- **Avançar**: executa o código passo a passo.
- **Entrar em**: executa o código passo a passo entrando em todos os métodos chamados.
- **Continuar**: continua a execução normal do programa (sem fazer o passo a passo).

Existe ainda a opção:

- **Terminar**: encerra a execução do programa.



Estratégias de Depuração: Resumindo...

Temos diferentes estratégias de depuração:

1. Ler (examinar) o código.
2. Teste de mesa (verificação manual passo a passo).
3. Explicar o código para alguém.
4. Inserir comandos de impressão.
5. Usar um depurador.

Qual a melhor estratégia?

A melhor estratégia depende de cada caso.

- ▶ Para um código recém-escrito ou para investigar apenas um segmento do programa, geralmente usamos a estratégia 1 ou 2.
- ▶ Muitas vezes usamos o depurador.
- ▶ Mas há situações em que não podemos usar o depurador. Por exemplo, quando o erro só ocorre no cliente, o uso de instruções de impressão (gerando logs de erro) pode ser a única saída.
- ▶ Lembre-se que explicar o código para alguém pode muitas vezes ser a forma mais fácil de identificar um erro.

Perguntas?

